VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky

Bakalářská práce

2013 **Ondřej Soukup**

VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra Informatiky

Návrh a implementace pokročilých stínovacích programů
Advanced shader design

2013 Ondřej Soukup

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání bakalářské práce

Student: **Ondřej Soukup**

Studijní program: B2647 Informační a komunikační technologie

Studijní obor: 2612R025 Informatika a výpočetní technika

Téma: Návrh a implementace pokročilých stínovacích programů
Advanced Shader Design

Zásady pro vypracování:

Cílem práce je návrh a implementace sady shaderů pro simulaci optických vlastností vybraných materiálů (zejména sklo, kovy, laky). Výsledkem by měla být implementace shaderů v jednom ze zvolených jazyků, ale také jejich detailní popis pro umožnění následné reimplementace těchto shaderů do libovolného prostředí.

1. Seznamte se s problematikou návrhu shaderů.
2. Zvolte vhodný jazyk a prostředí pro následnou implementaci (např. CgFX, FX Composer, CUDA-OptiX).
3. Navrhněte strukturu minimálně dvou shaderů ke všem uvedeným třídám materiálů.
4. Výsledný vzhled shaderů demonstrujte na vhodných jednoduchých modelech.

Seznam doporučené odborné literatury:

[1] ST-LAURENT, Sebastien. Shaders for Game Programmers and Artists. 2004. 512 s. ISBN 978-1592000920.
[2] ENGEL, Wolfgang. ShaderX7: Advanced Rendering Techniques. 2009. 773. ISBN 978-1584505983.
[3] http://developer.download.nvidia.com/shaderlibrary/webpages/cgfx_shaders.html

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí bakalářské práce: **Ing. Tomáš Fabián**

Datum zadání: 18.11.2011

Datum odevzdání: 07.05.2013

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Poděkování

Na tomto místě bych rád poděkoval mému vedoucím práce, Ing. Tomáši Fabiánovi, za vedení a pomoc při psaní této práce.

Prohlášení studenta

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě dne 7.5 2013

Podpis

Abstrakt

V současné době umožňuje stoupající výkon grafických karet používání stále pokročilejších efektů, díky kterým se počítačové simulace čím dál tím víc přibližují realitě. V této práci se snažím předvést několik postupů, jak s pomocí OpenGL a jazyka pro programování stínovacích programů GLSL, dosáhnout realisticky vypadající výsledků při simulaci vzhledu různých materiálů. Z důvodu nedostatku existujících aplikací vhodných pro vývoj stínovacích programů, také zběžně proberu způsob, jakým si napsat vlastní. V závěru potom srovnám dopad jednotlivých stínovacích programů na výkon.

Klíčová slova

OpenGL, GLSL, Vertex shader, Fragment Shader, BRDF

Abstract

Increasing performance of graphics cards currently allows use more advanced effects, due to which computer simulation becomes increasingly closer to reality. In this paper I try to present several methods using OpenGL and programming language GLSL shaders, achieve realistic-looking results when simulating the appearance of various materials. Due to the lack of existing applications suitable for the development of shaders, also briefly discuss how to write your own. In conclusion then compare the impact of different shaders on performance.

Key words

OpenGL, GLSL, Vertex shader, Fragment Shader, BRDF

Seznam použitých symbolů a zkratek

BRDF – Bidirectional Reflectance Distribution Function – Matematická funkce popisující vlastnosti povrchu. Existuje mnoho různých typů těchto funkcí, některé fyzikálně založené jiné čistě empirické.

Fragment – část obrazového výstupu z grafické karty, v některých případech se fragment rovná pixelu na monitoru.

Fragment shader – část stínovacího programu vykonaná nad každým fragmentem.

GLSL – zkratka pro OpenGL Shading Language, je součástí OpenGL

OpenGL – api pro vykreslování grafiky

Shader – stínovací program

Vertex – anglicky vrchol, základní stavební jednotka trojrozměrných těles

Vertex shader – část stínovacího programu prováděná nad vrcholy tělesa

Obsah

1 Úvod

Tato práce se zabývá programováním stínovacích programů v jazyce GLSL. V první kapitole projdeme dostupné programovací jazyky pro vývoj stínovacích programů a zdůvodním výběr OpenGL. Zběžně nastíním vývoj jednoduché aplikace pro testování a zobrazení vyvinutých stínovacích programů. Je to z toho důvodu, že v současné době není dostupná žádná aplikace, která by kompletně podporovala nejnovější specifikaci OpenGL. Pokud tedy chce člověk vyvíjet pro GLSL musí si napsat vlastní aplikaci.

V druhé kapitole probereme dělení materiálů z hlediska optických vlastností a dostaneme se k samotné implementaci stínovacích programů pro různé typy materiálů.

Na závěr srovnáme jednotlivé stínovací programy z hlediska výkonu a to na dvou grafických kartách.

2 Popis struktury shaderu a vývoje testovacího programu

2.1 Výběr jazyka pro implementaci

Ještě než se pustíme do programování shaderů je třeba si vybrat programovací jazyk a prostředí, ve kterém budeme shadery psát. Z jazyků se jsou na výběr momentálně tři.
Jsou to tyto:
HLSL
je jazyk vyvinutý společností Microsoft a je součástí DirectX od verze 8, syntaxe jazyka vychází z jazyka C.
GLSL
je jazyk pro programování shaderů, vyvinutý OpenGL ARB konsorcium jako součást OpenGL api. Jeho syntaxe vychází stejně jako HLSL z jazyka C.
Cg
Jazyk vyvinutý společností NVidia, syntaxe jazyka je opět založená na jazyku C. Dalo by se říct že je hierarchicky o něco víš než oba předchozí jazyky, protože kompiler jazyka Cg dokáže kompilovat jak do HLSL tak do GLSL.

Všechny výše zmíněné jazyky jsou si víceméně podobné a liší se jen v detailech, jako je pojmenování typů proměnných, jména funkcí a pár dalších drobností. Moje volba padla na OpenGL a to hlavně proto že se jedná o otevřený standard spravovaný konsorciem firem a ne o proprietární standard jako Cg a HLSL.

2.2 Vlastnost jazyka GLSL

Na začátek by asi bylo dobré zmínit pár vlastností jazyka GLSL. Projdeme jenom ty základní. Oproti standardním programovacím jazykům je zde mnoho funkcí zaměřených na grafiku.
Typy

Skalární

```
bool, uint, int, float , double
```

Vektorové

Všechny skalární typy mají svůj vektorový ekvivalent. Vektory mohou mít velikost od dvou do čtyř komponentů.

```
bvec2/3/4, uvec2/3/4, ivec2/3/4, vec2/3/4, dvec2/3/4
```

Na vektory lze použít stejné operátory jako na skalární typy. Operátory jsou použity po komponentech. Aby operátory fungovaly musí mít vektory stejnou velikost. K jednotlivým komponentům vektoru lze přistupovat pomocí tečky a písmena *x,y,z* nebo *w* k první, druhé, třetí a čtvrté komponentě vektoru například.

```
float temp = vektor.x;
```

Touto metodou lze přistupovat k až čtyřem komponentům a inicializovat tak například vektor o čtyřech komponentech pomocí vektoru se dvěma komponenty. Této technice se říká *swizzling*.

```
vec4 vektor4 =  vektor2.xxyx;
```

Nelze přistupovat ke komponentě w u vektoru o velikosti tři. Podobným způsobem jde zapisovat jen do konkrétních komponent vektoru, nebo na přeskáčku. Ovšem nejde zapsat při jedné operaci dvakrát do stejné komponenty jako v následujícím příkladě.

```
vektor3.xxy = vec3(1.0, 2.0, 3.0);
```

Tento zápis vypíše při kompilaci chybovou hlášku. Kromě *xyzw* lze použít ještě další dvě masky a to *rgba* pro barvy a *stpq* pro texturovací souřadnice, jedná se ale jenom o jiné pojmenování toho samého.

Matice

Dále jsou zde maticové typy. Všechny matice mají komponenty typu float nebo double. Matice jsou následujících typů.
`matmxn` - přičemž m a n mohou nabývat hodnot 2, 3 a 4 například
`mat3x4`
`mat4` je zkrácený zápis `mat4x4`
OpenGL používá column-major matice na rozdíl od DirectX, který používá row-major matice. K maticím nelze přistupovat pomocí swizzlingu, ale zato funguje syntaxe pro přístup k poli. Takže

```
mat4 matice;
matice[0] = vec4(1.0, 1.0, 1.0, 1.0);
```

nastaví první sloupec na vektor jedniček.

Opaque

Tyto typy reprezentují vnější objekt, jsou to reference na data. Jako takové mohou být použity jen jako argumenty funkcí, které z nich na základě dalších parametrů ta data získají. Mezi tyto typy patří především různé typy samplerů, tedy zjednodušeně řečeno textur.

Sampler

Samplerů je několik druhů, od jednorozměrné textury až po trojrozměrnou je tu *Sampler1D/2D/3D*, *SamplerCube* pro cubemap texturu dále jsou zde speciální typy pro pole výše zmíněných textur a ještě několik dalších, pro naše potřeby ale budou stačit tyto. Protože přístup k textuře není jenom triviální vyčtení hodnoty z pole, musí proběhnout filtrování a další procesy, používá se pro přístup k textuře funkce *texture()*. Tato univerzální funkce je GLSL implementovaná od verze 1.3, do této verze bylo potřeba používat pro přístup k různým typům textur různé funkce, například *textureCube()*.

2.3 Struktura shaderu

Vertex shader

```
#version 400 core
layout(location=0) in vec3 in_Position;
layout(location=1) in vec3 in_Normal;

out vec3 VertexView;
out vec3 NormalView;
out vec3 EyeVecView;
out vec3 LightVecView;

uniform vec3 LightPosition;
uniform vec3 EyePosition;
uniform mat4 ModelMatrix;
uniform mat4 ViewMatrix;
uniform mat4 ProjectionMatrix;
uniform mat4 NormalMatrix;

mat4 MV = ViewMatrix * ModelMatrix;
mat4 MVP = ProjectionMatrix * ViewMatrix * ModelMatrix;

void main(void)
{
   gl_Position = MVP * vec4(in_Position, 1.0);
   VertexView = vec3(MV * vec4(in_Position, 1.0));
   NormalView = (normalize(NormalMatrix  * vec4(in_Normal,0.0))).xyz;
   vec3 LightPosView = (MV * vec4(LightPosition, 1.0)).xyz;
   EyeVecView = normalize(-VertexView);
   LightVecView = normalize(LightPosView - VertexView);
}
```

Vertex shader zpracovává jednotlivé vrcholy 3D modelu, jeden po druhém a neví nic o ostatních vrcholech. Pro každý příchozí vrchol, vrátí shader právě jeden transformovaný vrchol. Každý vrchol má sadu uživatelsky definovaných parametrů, například normálu, pozici nebo texturovací souřadnice.

Struktura shaderu se od normálního programu napsaného v C moc neliší. Na prvním řádku je direktiva pro preprocesor s požadovanou verzí do které je třeba shader zkompilovat. Dále jsou to vstupní parametry označené klíčovým slovem *in* a typem proměnné. Klíčové slovo out označuje výstupní proměnnou, která bude předaná do fragment shaderu. Proměnné typu *uniform* jsou parametry shaderu, které jsou dodávaný za běhu z programu, o tom jak je shaderu předat si povíme o kus dál. V shaderu můžeme deklarovat funkce stejně jako v jiném programu, ty můžou buď něco vracet nebo nevracet nic. Vstupním bodem programu je jako v C funkce *main*. Ve vertex shaderu probíhá transformace příchozích vrcholů a jejich odeslání do fragment shaderu pomocí vestavěné proměnné *gl_Position*. Výše uvedený shader bude s mírnými úpravami základem všech následujících shaderů.

Fragment Shader

```
#version 400

in vec3 NormalView;
in vec3 EyeVecView;
in vec3 LightVecView;

out vec4 FragColor;

uniform vec3 ModelColorPhong;
uniform vec3 LightColor;

void main(void)
{
   vec3 N = normalize(NormalView);
   vec3 L = normalize(LightVecView);
   vec3 E = normalize(EyeVecView);
   vec3 H = normalize(E + L);
   float NdotL = max(dot(N, L), 0.0);
   float NdotH = max(dot(N, H), 0.0);
   //half lambert
   float D = pow((NdotL * 0.5 + 0.5),2.0);
   vec4 Diffuse = vec4(ModelColorPhong * D, 0.0);
   vec4 Specular = vec4(LightColor * pow(NdotH, 50), 0.0);

   FragColor = Diffuse + Specular;
}
```

Ve fragment shaderu jsou zpracovávány jednotlivé části obrazu, dalo by se říct jednotlivé pixely, i když to není vždy úplně pravda. Pro fragment shader jsou přístupná v interpolovaná data vypočtená při rasterizaci, jeho pozice na obrazovce a uniform proměnné. Kromě toho jsou tu ještě vestavěné proměnné, mimo jiných to je *gl_FrontFacing*, *gl_ClipDistance*, *gl_FragCoord*. Podobně jako vertex shader ani fragment shader nemá přístup k jiným datům než k těm svým. Fragment shader uvedený výše je mírně upravený Blinn-Phongův stínovací model. Rozdíl je ve vynechání ambientní složky osvětlení, která je nahrazená Lambertovým modelem převedeným z rozsahu -1 – 1 do 0 - 1 čímž vzniknou jemnější přechody mezi světlem a stínem.

Geometry shader

Tahle programovatelná část grafické pipeline pracuje s jednotlivými primitivy, ze kterých je složený modelu. Má přístup ke všem informacím týkajících se jednotlivého primitiva, například u trojúhelníku jsou to všechny jeho vrcholy. Pokud to specifikujeme tak má přístup i k přilehlým vrcholům. V geometry shaderu lze na rozdíl od vertex shaderu ubírat nebo přidávat vrcholy a tím ovlivňovat výsledný tvar tělesa. Můžeme ho použít ke generování jednoduché vegetace, nebo k přidání detailů na modelu. Není

Tessellation shader

Tento shader jsou ve skutečnosti shadery dva, Tessellation control shader a Tessellation evaluation shader. Tyto shadery umožňují podobně jako geometry shader měnit geometrii objektu, ovšem jejich podpora je zabudovaná přímo v grafickém hardwaru a mohou přidávat mnohem větší množství geometrie. Mimo jiné se dají využít na adaptivní nastavení detailů objektů na základě vzdálenosti od kamery.

2.4 Program pro zobrazení shaderů

Už víme jak napsat shader, ale nevíme kde si prohlédnout co vlastně dělá. Jednou možností je stáhnout si nějaké prostředí pro vývoj shaderů a víc neřešit. Možností je několik například **Shader Designer**, **RenderMonkey**, **FX Composer** nebo **glslDevil**. Ale kromě programu FX Composer chybí všem podpora novějších verzí GLSL a jejich vývoj byl ukončen. FX composer zase nepodporuje GLSL vůbec. Nezbývá nám tedy než si napsat program vlastní.

Vytvořit ve Windows OpenGL okno není s pomocí knihoven glew a freeglut je poměrně jednoduché a proto se o tom nebudu příliš rozepisovat. O čem bych se ale zmínil, jsou novinky, které do OpenGL přibyly s verzí 3.0. Aktuální verze je sice 4.3 ale nejvíce inovací přinesla právě verze 3.0. Za prvé byl označený jako zastaralý takzvaný immediate mód, tedy příkazy glBegin, glEnd spolu s glVertex. Nebo glDrawArrays a glDrawElements s glEnableClientState. Je to především proto, že v tomto módu byla neúměrně zatěžována komunikační sběrnice mezi procesorem a grafickou kartou neustálým tokem dat a to mělo neblahý vliv na výkon. Od verze 3.0 je se jako informace používají generické atributy spolu s vertex buffer objekty. To přináší na jednu stranu velkou míru svobody na druhou stranu vyžaduje učení něčeho nového od těch co jsou zvyklí na starý způsob.

Další zásadní změnou v OpenGL je odstranění původního zásobníku matic a všech funkcí s ním spojených, například glPushMatrix, glPopMatrix, glProjection, glTranslate, glRotate a glScale. To nás nutí napsat si vlastní funkce pro vytváření matic a práci s nimi, nebo se podívat po vhodné knihovně. Jednou takovou velmi povedenou knihovnou je GLM (OpenGL Mathematics), kterou napsal Christophe Riccio. Knihovna obsahuje funkce, které dokáží nahradit výše zmíněné zastaralé funkce. Navíc je design knihovny podobný specifikacím GLSL, takže má podobnou i syntaxi, což ji dělá velmi lehce použitelnou. Knihovnu lez stáhnout ze stránek *http://glm.g-truc.net*. Další výhodou je že knihovna je „header only“ takže není třeba při kompilaci linkovat knihovny, stačí přidat hlavičkové soubory.

Kompilace shaderu

Kompiler pro GLSL je zabudovaný do OpenGL knihovny a shadery mohou být zkompilovány jen v kontextu už běžícího OpenGL programu. Až do verze 4.1 nebylo možné zkompilované shadery nijak uložit. Postup při kompilaci je následující. Nejdříve požádáme OpenGL o vytvoření shaderu pomocí funkce *glCreateShader* a jako parametr předáme požadovaný typ shaderu (*GL_FRAGMENT_SHADER, GL_VERTEX_SHADER*). Funkce vrací ukazatel.

```
GLuint vertShader = glCreateShader(GL_VERTEX_SHADER);
```

Vytvořené místo pro shader je ještě potřeba naplnit kódem shaderu, pomocí funkce glShaderSource. Ta přijímá za parametry ukazatel, který jsme získaly v předchozím kroku, počet shaderů ke kompilaci, pole znaků se zdrojovým kódem a jeho délku. Pokud jako délku zadáme null musí být řetězec ukončený znakem null. Takto vytvoříme objekty pro vertex a fragment shader. Část shaderu ještě zkompilujeme pomocí *glCompileShader*.

```
glShaderSource(vertShader, 1, sourceArray, NULL);
glCompileShader();
```

Nyní je třeba vytvořit z obou částí kompletní program. Nejdřív vytvoříme program pomocí funkce *glCreateProgram*, která vrátí ukazatel. K vytvořenému programu nyní připojíme dříve vytvořené vertex a fragment části pomocí příkazu *glAttachShader*, jako parametry předáme ukazatel

na program a připojovanou část. Nakonec celý shader program zkompilujeme příkazem *glLinkProgram*.

```
GLuint program = glCreateProgram();
glAttachShader(program, vertShader);
glAttachShader(program, fragShader);
glLinkProgram(program);
```

V jednotlivých fázích kompilace a linkování programu je žádoucí ověřovat stav a hlásit případné chyby při kompilaci. To můžeme pomocí funkce *glGetShaderiv* pro vertex a fragment program a *glGetProgramiv* pro kompletní shader. Program do grafické pipeline vložíme příkazem *glUseProgram*.

```
glUseProgram(program);
```

Zjištění uniform proměnných shaderu

Tento krok je potřeba ke zjištění jmen, typu a hlavně umístění proměnných. Bez něj bychom nemohli předávat shaderu vstupní parametry, transformační matice ani textury. Postup je následující. Nejprve pomocí *glGetProgramiv* zjistíme počet uniform proměnných v shaderu a délku nejdelšího názvu. Alokujeme pole pro název a v cyklu můžeme zjistit pomocí *glGetActiveUniform* jména všech proměnných a jejich typ. Pak pomocí funkce *glGetUniformLocation*, která jako jeden z parametrů vyžaduje jméno proměnné zjistíme její umístění. Umístění proměnné potřebujeme, abychom ji byli schopni shaderu dodat pomocí funkce *glUniform*.

```
GLint maxLength, nAttribs;
glGetProgramiv(_programHandle, GL_ACTIVE_UNIFORM_MAX_LENGTH, &maxLength);
glGetProgramiv(_programHandle, GL_ACTIVE_UNIFORMS, &nAttribs);
GLchar *name = new GLchar[maxLength];
GLint written, size, location;
GLenum type;

for(int i = 0; i < nAttribs; i++)
{
	glGetActiveUniform(_programHandle, i, maxLength, &written, &size, &type, name);
	location = glGetUniformLocation(_programHandle, name);
}
```

Vykreslení objektu

Je několik způsobů jak dát OpenGL povel k vykreslení objektu a s nimi souvisí i různé způsoby posílání per-vertex dat.

Immediate mode

Je zastaralý od verze 3.0, každý typ dat má svou cestu, například souřadnice vrcholů jsou posílány pomocí *glVertex3f*, normály *glNormal3f* a v shaderu k nim přistupujeme pomocí proměnných *gl_Vertex* a *gl_Normal*.

DrawArrays

Data jsou nejprve nahrána v poli do paměti grafické karty a pak sekvenčně procházená a vykreslovaná. Tento způsob mnohem méně zatěžuje paměťovou sběrnici. Jednu nevýhodu ovšem má. Protože jsou data procházena sekvenčně vyskytují se zde některé vrcholy vícekrát, například pro vykreslení jednoho čtverce je potřeba šesti vrcholů, dva z vrcholů jsou duplicitní. Dochází k velkému plýtvání pamětí.

DrawElements

Je asi nejúspornější metoda vykreslování z výše zmíněných metod. Používá stejně jako metoda DrawArrays pole vrcholů, ale na rozdíl od ní je každý vertex v poli jen jednou a pro sestavování trojúhelníků je použité pole indexů.
Tuto metodu jsem použil i ve svém programu a proto bych se u ní zastavil a popsal jí trochu detailněji.
Než začneme vytvářet vertex buffer objekty je dobré mít k dispozici data 3d objektu (pozice vrcholů, normály, tangenty, texturovací souřadnice). Já jsem pro tento účel využil služeb knihovny assimp [http://assimp.sourceforge.net/]. Způsob jakým vytváříme vertex arrays je velmi podobný vytváření shader programu. Nejdřív požádáme OpenGL o vytvoření vertex array příkazem glGenVertexArrays, které pak následně nastavíme jako aktivní. Tím pádem budou následující příkazy prováděny v něm.

```
GLuint vaoHandle;
glGenVertexArrays(1, &vaoHandle);
glBindVertexArray(vaoHandle);
```

První buffer, který vygenerujeme bude obsahovat indexy jednotlivých vrcholů. Příkaz *glGenBuffers* vygeneruje buffer, který příkazem glBindBuffer nastavíme jako aktuální a příkazem *glBufferData* naplníme.

```
GLuint indBuff;
glGenBuffers(1, indBuff);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, indBuff);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, (3 * numTriangles) * sizeof(GLuint), indBuffArray,
GL_STATIC_DRAW);
```

Pak následují buffery kde jsou pozice vrcholů, normály nebo texturovací souřadnice. Postup je stejný jako u bufferu s indexy vrcholů, akorát místo konstanty *GL_ELEMENT_ARRAY_BUFFER* uvedeme *GL_ARRAY_BUFFER*. Po nahrání dat do bufferu je třeba povolit použití tohoto pole atributů pomocí *glEnableVertexAttribArray*. Tato funkce vyžaduje jako argument index generického atributu k aktivaci. Nakonec nastavíme parametry atributu funkcí *glVertexAttribPointer*. Je třeba nastavit typ parametru a jeho velikost.

```
GLuint vertBuff;
glGenBuffers(1, &vertBuff);
glBindBuffer(GL_ARRAY_BUFFER, vertBuff);
glBufferData(GL_ARRAY_BUFFER, numVerttices * 3 * sizeof(float), vertices, GL_STATIC_DRAW);
glEnableVertexAttribArray(index);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FLASE, 0, 0);
```

U předchozího kroku bych se ještě zastavil. Jsou totiž dva způsoby, jak uložit data do array bufferu. Za prvé je možné vytvořit pro každý atribut (poloha, normála, tangenta) vlastní buffer. Tahle možnost je o něco jednodušší, při volání funkce *glVertexAttribPointer* protože není třeba uvádět poslední tři parametry. Není to ovšem ideální řešení z hlediska výkonu.

Druhou možností je uložit vše do jednoho pole za sebou. Jako na následujícím schématu.

|----------vertex 1----------||----------vertex 2 ----------||
| pozice | normála | tex. souřadnice || pozice | normála | tex. souřadnice ||

Nyní máme tři atributy v jednom poli a musíme zavolat funkci *glVertexAttribPointer* třikrát na stejný buffer.

```
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(GLfloat) * 8, 0);
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, sizeof(GLfloat) * 8, sizeof(GLfloat) * 3);
glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, sizeof(GLfloat) * 8, sizeof(GLfloat) * 6);
```

Ke změně došlo jen u posledních dvou argumentů, které jsou stride a offset. Stride je celková velikost dat pro jeden vertex a offset je odstup stávajícího atributu od prvního.

V shaderu můžeme k atributům přistupovat opět několika způsoby. U obou způsobů jsou atributy označeny klíčovým slovem in, například *in vec3 position;*.
První způsob je svázat atribut na určité pozici se jménem atributu v shaderu funkcí *glBindAttribLocation*.
Druhou možností je určit v shaderu na jaké pozici se daný atribut nachází pomocí klíčového slova layout takto: *layout (location = 0) in vec3 position*.

3 Popis stínovacích programů

3.1 Dělení materiálů podle optických vlastností

Protože reakce fotonů s povrchem materiálu úzce souvisí s tím, jestli je daný materiál schopný vést elektrický proud můžeme materiály rozdělit do dvou základních skupin na dielektrické a kovové. Pro dielektrické materiály jsou charakteristické tyto vlastnosti. Jejich interakce se světlem je minimální a proto některé světlo propouští a jsou průhledné, při odrazu odráží celé spektrum se stejnou intenzitou. Do této skupiny patří sklo, plasty, dřevo.

Kovy jsou vodivé neprůhledné materiály s vysokou odrazivostí. Některé kovy neodráží celé spektrum se stejnou intenzitou zbarvují tak odrazy.

Složené materiály jako například metalický lak kombinuje vlastnosti obou předešlých skupin. V tenké vrstvě dielektrického laku jsou rozptýlené částečky kovu.

3.2 Dielektrické materiály

Sklo

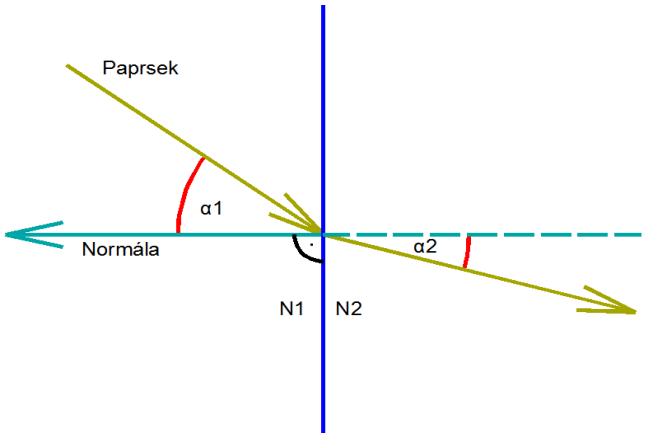
Sklo patří mezi dielektrické materiály. To je skupina materiálů, které nevedou elektrický proud. Mají relativně malou odrazivost, která jen minimálně závislá na barvě osvětlení. Díky jejich malé interakci se světlem jsou některé dielektrika průhledné. Mezi tyto materiály patří plasty, porcelán, sklo. Světlo při průchodu těmito materiály zpomaluje, čehož důsledkem je jeho lom.

Odraz, lom a rozklad což jsou hlavní vlastnosti skla, které budeme simulovat se dají popsat pomocí geometrické optiky a to je dobře protože právě na tuto část optiky je grafický hardware zařízený nejlépe. Interference, ohyb nebo polarizace což jsou jevy popisované pomocí vlnové optiky, nemají pro uvěřitelnou simulaci skla velký význam. Jevy popisované kvantovou optikou jsou pouhým okem většinou nepozorovatelné a tudíž pro nás zanedbatelné.

Lom

$$\frac{\sin(\alpha 1)}{\sin(\alpha 2)} = \frac{N1}{N2}$$

Snellův zákon



Lom paprsku

Vztah mezi dopadajícím paprskem a paprskem lomeným popisuje Snellův zákon.

Ten popisuje úhel dopadu (*α1*), což je úhel mezi dopadajícím paprskem a normálou povrchu, úhel lomu (*α2*) jako úhel mezi procházejícím paprskem a prodlouženou normálou. Materiály jsou popsány pomocí indexu lomu N1 a N2. Poměr mezi dvěma indexy lomu určuje velikost ohybu paprsku na rozhraní látek. Index lomu se pro různé typy skel pohybuje v rozmezí od 1,5 do 1,9.

S tímto vzorcem a trochou matematického úsilí by se dala napsat funkce, která z vektoru dopadajícího na povrch tělesa udělá vektor lomový. My ovšem máme štěstí, protože přesně taková funkce je v GLSL definovaná a jmenuje se *refract*. Ta ze vstupních parametrů, kterými jsou vektor dopadu I, normála N a poměr indexů lomu ETA, vypočítá vektor lomu. V GLSL je funkce už od verze 1.10 a je definovaná takto:

```
k = 1.0 - ETA * ETA * (1.0 - dot(N, I) * dot(N, I));
if (k < 0.0)
    R = genType(0.0);
else
    R = ETA * I - (ETA * dot(N, I) + sqrt(k)) * N;
```

Odraz

U většiny průhledných objektů ovšem není všechno světlo propuštěno ale jeho část je odražena. Čím je úhel dopadu větší tím je poměr odraženého a propuštěného světla větší. Při úhlu větším než je mezní úhel nastává totální odraz, což je situace kdy není žádné světlo propuštěno a je celé odraženo. To nastává pokud se paprsky šíří z opticky hustšího prostředí a úhel lomu je roven pravému úhlu. V tomto případě je *sinα2 = 1* a zákon lomu má tvar *sinαm = sinα1 = N2/N1*. Mezní úhel jde spočítat ze vztahu *sinαm = arcsin(N2/N1)*. Na obrázku je tento jev patrný poblíž pravé hrany objektu. Tento efekt lze velmi věrně napodobit pomocí Fresnelovy funkce, která vypadá následovně.

$$R_\mathrm{s} = \left|\frac{n_1\cos\theta_\mathrm{i} - n_2\cos\theta_\mathrm{t}}{n_1\cos\theta_\mathrm{i} + n_2\cos\theta_\mathrm{t}}\right|^2 = \left|\frac{n_1\cos\theta_\mathrm{i} - n_2\sqrt{1-\left(\frac{n_1}{n_2}\sin\theta_\mathrm{i}\right)^2}}{n_1\cos\theta_\mathrm{i} + n_2\sqrt{1-\left(\frac{n_1}{n_2}\sin\theta_\mathrm{i}\right)^2}}\right|^2$$

$$R_\mathrm{p} = \left|\frac{n_1\cos\theta_\mathrm{t} - n_2\cos\theta_\mathrm{i}}{n_1\cos\theta_\mathrm{t} + n_2\cos\theta_\mathrm{i}}\right|^2 = \left|\frac{n_1\sqrt{1-\left(\frac{n_1}{n_2}\sin\theta_\mathrm{i}\right)^2} - n_2\cos\theta_\mathrm{i}}{n_1\sqrt{1-\left(\frac{n_1}{n_2}\sin\theta_\mathrm{i}\right)^2} + n_2\cos\theta_\mathrm{i}}\right|^2$$

$$R = \frac{R_s + R_p}{2}$$

Popisuje množství světla, které je odraženo jako funkci úhlu dopadu, polarizace světla a poměru indexů lomu. Rs vyjadřuje část odraženého světla, která je s-polarizovaná, to znamená že jeho elektrické pole je v rovině s odrazovou plochou. Druhá forma rovnice je rozepsaná pomocí Snellova zákona a trigonometrických funkcí. Rp je potom druhá část, jehož polarizace je kolmá na polarizaci Rs. Odražené světlo je nepolarizované a obsahuje stejnou část Rp jako Rs. Tato rovnice je pro real-time shader výpočetně náročná a proto v shaderu použijeme její zjednodušení.

Rozklad

Dalším důsledkem přechodu mezi materiály je chromatická aberace, neboli rozklad světla na jednotlivé barevné složky. Jde o to, že různé barevné složky světla se lámou každá pod trochu jiným úhlem a to má za následek rozklad bílého světla na jednotlivé barevné složky. Tento jev je patrný především u silnějších skleněných objektů, jako silné lupy, skleněné sošky a podobně. I tento jev je celkem jednoduché simulovat pomocí GLSL. Stačí funkci refract předat pro každou ze tří barevných složek mírně odlišný index lomu.

Implementace

Vertex Shader

Ve vertex shaderu probíhají ty méně zajímavé, nicméně pro běh shaderu nepostradatelné, procesy. Pro výpočet osvětlení ve fragment shaderu budeme potřebovat vektory světla a pohledu, pozici právě zpracovávaného vrcholu a normálu. Všechno samozřejmě v pohledových souřadnicích. Pro výpočet odrazu a lomu světla ještě musíme vypočítat normálu a vektor pohledu ve světových souřadnicích. Kromě povinné transformace vrcholu do projekčních souřadnic se zde již nic zajímavého neděje.

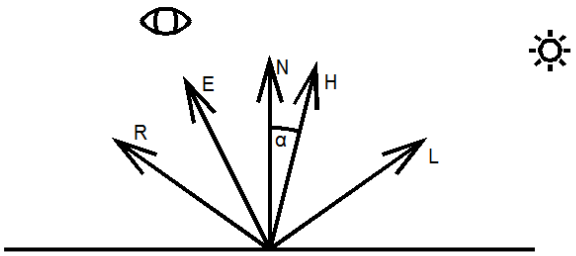
Fragment Shader

Všechny výpočty osvětlení se odehrávají právě tady. Vezmeme to popořadě.

```
float Fresnel(vec3 N, vec3 E)
{
  float fr =  1.0 - dot(E, N);
  float exp = pow(fr, 3.0);
  fr = exp + 0.01 * (1.0 – exp);
  return fr;
}
```

Funkce Fresnel je zjednodušením Fresnelovy rovnice podle Schlicka. Tato funkce má velmi podobný průběh jako originální Fresnelovy rovnice pro materiály, které mají imaginární část indexu lomu rovnu nule, protože tento argument je ve zjednodušené funkci zanedbán. Pro materiály jako kovy, které mají imaginární část větší se už není tato funkce moc přesná. Vstupními parametry jsou Normála N, vektor pohledu E. Vrací poměr mezi prošlými a odraženými paprsky.

```
vec4 SpecularReflection(vec3 E, vec3 L, vec3 N, vec3 Color, float Rp)
{
  vec3 H = normalize(E + L);
  vec4 Spec = vec4(Color * pow(max(dot(H, N),0.0),Rp), 0.0);

  return Spec;
}
```



Vektory paprsků

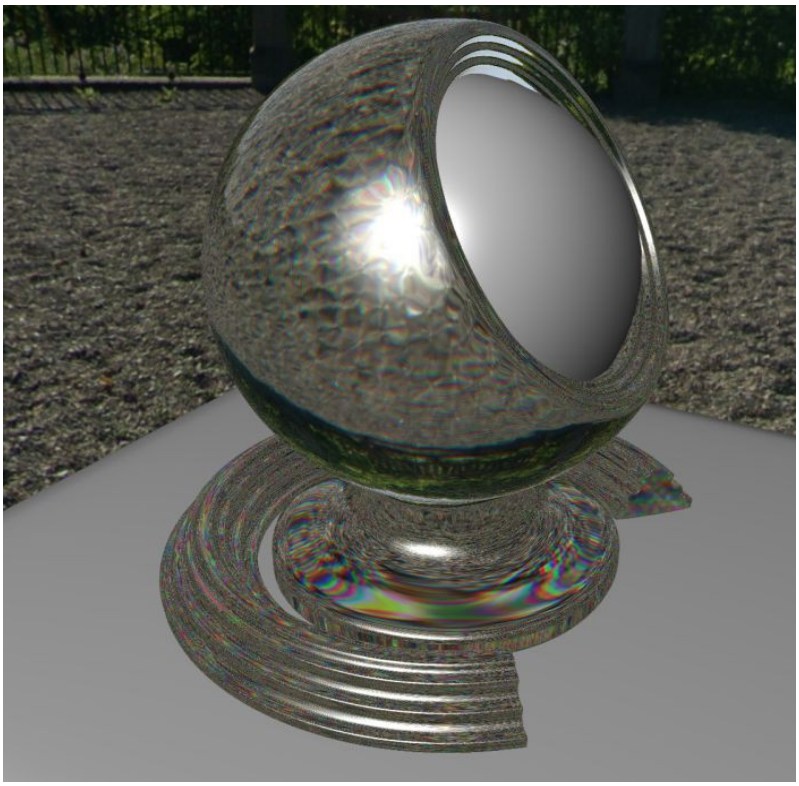
SpecularReflection je funkce pro výpočet spekulárních odlesků podle Blinn-Phongova stínovacího modelu. Klasický Phongův model požívá pro výpočet síly spekulárního odrazu skalární součet vektoru světla odraženého od povrchu objektu R a vektoru k pozorovateli E. Blinn-phongův model využívá skalárního součinu normály N a vektoru H, což je normalizovaný součet vektoru L a E. To je výhodné především z hlediska výkonu, protože součet a normalizace dvou vektorů je výpočetně mnohem méně náročná než výpočet odraženého vektoru. Aby byl výsledek podobný, je třeba výsledek umocnit na číslo přibližně čtyřikrát větší větší než v předchozím případě.

```
void main(void)
{
  vec3 Ew = normalize(EyeVecWorld);
  vec3 E = normalize(EyeVecView);
  vec3 N = normalize(NormalView);
  vec3 L = normalize(LightVecView);
  vec3 Nw = normalize(NormalWorld);
  vec4 ReflectColor = texture(CubeMapTex, reflect(-Ew, Nw));
  vec4 RefractColor = vec4(0.0, 0.0, 0.0, 0.0);
  RefractColor.x = texture(CubeMapTex, refract(-Ew, Nw, RefractionIndex - 0.005)).r;
  RefractColor.y = texture(CubeMapTex, refract(-Ew, Nw, RefractionIndex)).g;
  RefractColor.z = texture(CubeMapTex, refract(-Ew, Nw, RefractionIndex + 0.005)).b;
  float F = Fresnel(N, E);
  vec4 SpecColor = SpecularReflection(E, L, N, LightColor, ReflectionPower);

  out_Color = mix(RefractColor, ReflectColor, F) + SpecColor;
}
```

Ve funkci main nejdříve normalizujeme vektory poslané z vertex shaderu kvůli jejich správné interpolaci a přesnějšímu výpočtu per-pixel osvětlení. Odraženou barvu okolí získáme z tzv. cubemap textury, což je textura složená ze šesti částí a namapovaná na krychli. Jako texturovací souřadnice použijeme vektor vrácený funkcí reflect. Barvu průhledné složky získáme velmi podobně také z cubemap textury, ale jako texturovací souřadnice tentokrát požijeme vektor vrácený funkcí refract. Tato funkce potřebuje jako vstupní parametry vektor k pozorovateli, normálu a index lomu. Pro každou ze tří složek barvy použijeme index lomu mírně odlišný.

Výslednou barvu fragmentu získáme kombinací odražené a průhledné složky pomocí funkce mix, která provádí lineární interpolaci mezi mezními hodnotami, v našem případě barvami. A přičteme barvu spekulární složky osvětlení.

Výsledek



Shader - Sklo

Závěr

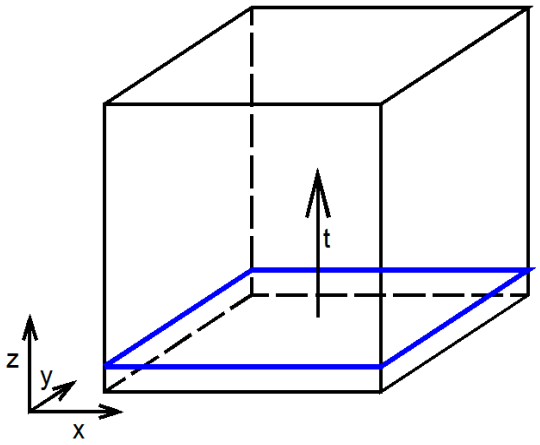
Tato implementace shaderu pro sklo vytváří na první pohled docela realistický dojem skla, má ovšem několik nedostatků. Asi největším problémem je, že ostatní objekty ve scéně nejsou skrz sklo vidět a ani se v něm neodráží. To je patrné i na obrázku, objekt šedivé koule uprostřed není vidět, přesto že by z tohoto úhlu pohledu měl být vidět. Toto by asi šlo vyřešit dvou průchodovým renderováním, kdy v prvním průchodu by se vyrenderovala scéna bez objektů se skleněným shaderem do textury a ta by se v druhém průchodu použila místo cubemap pro výpočet průhlednosti a odrazů. Další věcí která sklu chybí je kaustika, což jsou oblasti na podkladu do kterých se soustředí lomené paprsky světla procházejícího sklem.

Voda

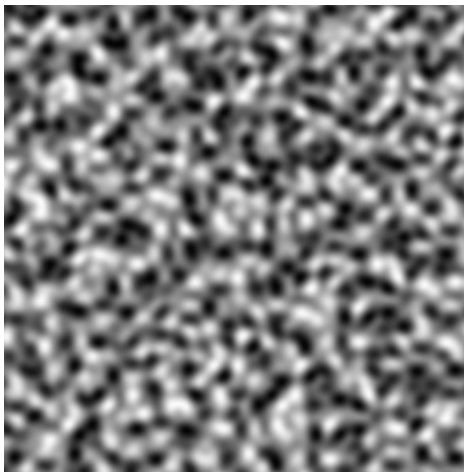
Přesto že je voda poměrně odlišná látka od skla, jsou jejich optické vlastnosti velmi podobné. Stejně jako sklo i voda část paprsků láme, část odrazí. Díky tomu lze pro simulaci klidné vodní hladiny využít velmi podobný shader jako pro sklo. Problém je že vodní hladina bývá velmi zřídka klidná. Vlivem i slabého vánku na hladině vzniknou vlnky. Pro statický obraz by stačila normálová mapa vlnitého povrchu, ale pro použití ve hře je třeba nalézt jiné řešení. Tím je Simplex šum generovaný v reálném čase a metoda bump mapping.

Simplex vychází z Perlinova šumu, ale má několik odlišností. Především je méně náročný na výpočet, lépe škáluje do vyšších dimenzí s menší výpočetní náročností, nemá zřetelné směrové artefakty. Simplex šum patří do skupiny gradientních šumů. To znamená že je tvořen z mřížky náhodných gradientů, které jsou mezi sebou interpolovány pro získání hodnot mezi mřížkou.

Simplex šum lze vytvářet v různých dimenzích, pro naše potřeby bude nejlepší použít funkci pro generování trojrozměrného šumu. A to z toho důvodu, že potřebujeme dosáhnout vlnění povrchu vody, které se v čase mění. To provedeme tak, že v každém vykresleném rámci použijeme dvourozměrný průřez krychle a v každém dalším se v pomyslné krychli posuneme o kousek výš. Díky tomu, že simplex je spojitá funkce nebude docházet k žádným skokovým změnám v textuře ale přechody budou plynulé.



3D textura šumu a a její 2D průřez



Simplex šum

Samotná textura je jenom černobílý šum a je třeba ho nějak převést na povrchové nerovnosti. Přesně toho dosáhneme technikou zvanou bump mapping. Ta vytváří nerovností na jinak rovném povrchu modelu pomocí výškové mapy. To je černobílá textura, kde světlá místa znázorňují místa položená výš než tmavá místa. Oproti normal mappingu je tato metoda méně přesná a flexibilní, ale pro naše potřeby naprosto dostačující.

Jinou metodou pro využití černobílé šumové textury k simulaci vody by bylo použít texturu už ve vertex shaderu k posunutí jednotlivých vrcholů modelu takzvaný displacement mapping, tím že by se funkce pro výpočet textury volala už ve vertex shaderu by došlo k výraznému snížení hardwarových nároků. Zároveň by však bylo třeba použít těleso s vysokým počtem vrcholů pro dosažení stejné kvality.

Implementace

Vertex shader

Do vertex shaderu jdou jako vstupní informace poloha vrcholu, jeho normála, tangenta, bitangenta a texturovací souřadnice. Normálu, tangentu a bitangentu potřebujeme na sestavení TBN matice pro převod z tangentového do pohledového prostoru, kde se počítá osvětlení. Dále zde proběhne klasické sestavení vektorů pohledu a světla a jejich převod do pohledových souřadnic. Také zde sestavíme trojrozměrné texturovací souřadnice pro použití při generování textury šumu. Sestavíme je z příchozích texturovacích souřadnic a jako souřadnici Z použijeme uniformní proměnou Time dodanou programem. Proměnná Time se mění před začátkem vykreslování snímku, aby nedošlo k nesouvislému vykreslení textury. Nakonec zapíšeme do povinné proměnné gl_Position.

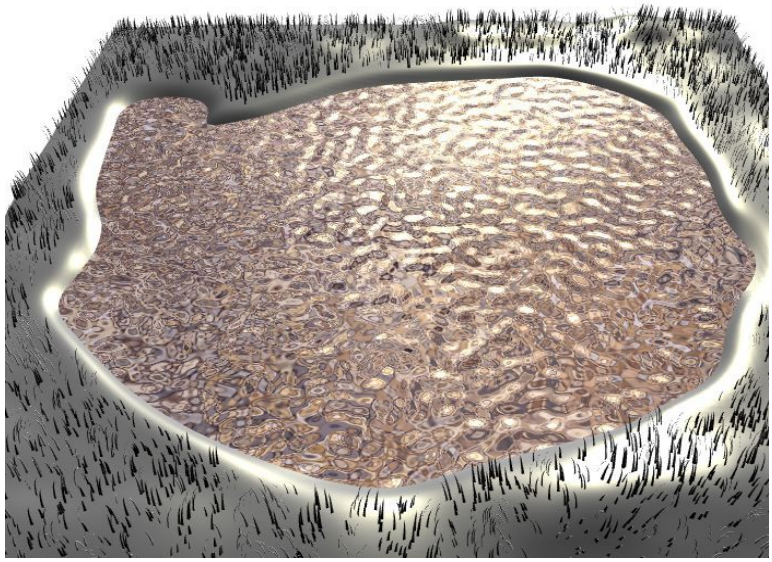
Fragment shader

Větší část shaderu zabírá kód pro generování trojrozměrného simplex šumu, funkce snoise. Tuto část kódu jsem převzal z práce Iana McEwana, který přišel se způsobem to jak implementovat simplex šum přímo v shaderu bez nutnosti použití textury gradientu pro jeho výpočet. GLSL sice má definovanou funkci *noise1/2/3/4()*, ale ne každý hardware jí podporuje, konkrétně na kartách NVidia vrací tato funkce vždy nula, to jsem ověřil na kartě NVidia GT 130M a GT660M, ani karta Intel HD Graphics 4000 ji nemá implementovanou. Karty ATI/AMD by sice tuto funkci podporovat měli, ale to jsem neměl na možnost otestovat. Nicméně je jasné že na podporu této funkce na hardwaru není možné se spolehnout a je proto lepší použít vlastní.

```
void main( void )
{
  vec3 E = normalize(EyeVecView);
  vec3 Ew = normalize(EyeVecWorld);
  vec3 L = normalize(LightVecView);
  vec3 H = normalize(E + L);

  float a = snoise(UV_3D);
  float b = snoise(UV_3D + vec3(Offset, 0.0, 0.0));
  float c = snoise(UV_3D + vec3(0.0, Offset, 0.0));

  vec3 N = vec3(b-a,c-a,1.0);

  N = normalize( outTBN * N );

  vec3 Nw = MV3x3 * N;

  vec4 LookUpColor = texture(CubeMapTex, refract(-Ew, Nw, RefractionIndex ));

  float s = pow(max(dot(H, N), 0.0), ReflectionPower);

  vec4 Color = LookUpColor + vec4(LightColor, 0.0) * s + vec4(WaterColor, 0.0);

  FragColor = Color;
}
```

Ve funkci main normalizujeme vektory vypočtené ve vertex shaderu. Pak vygenerujeme tři hodnoty šumu, jednu na souřadnicích *UV_3D*, jednu posunutou po ose X o hodnotu Offset a poslední posunutou po ose Y opět o Offset. Tím získáme tři odlišné hodnoty, ze kterých spočteme dva gradienty a ty použijeme jako x a y komponenty normály, z bude jedna. Tím jsme získali normálu zvlněného povrchu v tangentových souřadnicích. Teď je třeba jí převést do pohledových souřadnic pro výpočet osvětlení a do světových souřadnic pro výpočet průhlednosti. Pak už jen získáme barvu průhlednosti z cubemap textury a vypočteme spekulární odlesky.

Výsledek



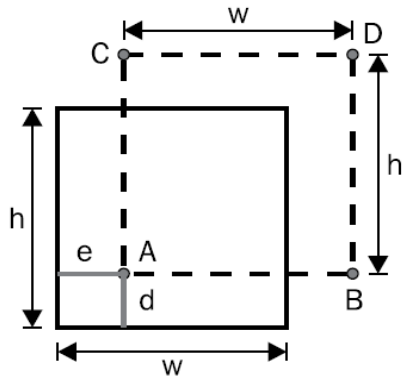
Shader - Voda

Závěr

Nesnažil jsem se o žádnou fyzikálně věrnou simulaci chování vody pouze o uvěřitelnou simulaci, což se myslím povedlo. Stejně jako shader skla má i voda několik neduhů, tím hlavním je jako u skla jenom částečná průhlednost.

Mramor

Další materiál, který patří k mezi dielektrické je mramor. Hlavním optickým rozlišovacím prvkem mramoru je jeho textura. Ta je velmi charakteristická a podle mě je pro uvěřitelnou simulaci mramoru zásadní. Proto jsem se rozhodl zaměřit hlavně na ní. Mramoru má mnoho různých typů, od černé po čistě bílý, ale většina typů má jedno společné a to je žilkovitá struktura ne nepodobná letokruhům ve dřevu, ovšem mnohem více nahodilá. K napodobení nahodilosti nám pomůže Perlinův šum, ale tentokrát jeho jeho trojrozměrná varianta. Podobně by posloužila i dvourozměrná textura, ale u dvourozměrné textury je potřeba správně namapovat texturovací souřadnice a udělat to tak aby nebyly vidět švy a měřítko textury bylo po celém povrchu přibližně stejné. U trojrozměrné textury takové starosti odpadají. Je ovšem třeba udělat trojrozměrnou texturu bezešvou. Je několik způsobů jak toho dosáhnout. Já si vybral cestu lineární interpolace hodnot vrácených šumovou funkcí. Výsledek je sice na okrajích trochu rozpitý, ale pro naše účely poslouží dobře. Jak to funguje.



Interpolace bezešvé textury

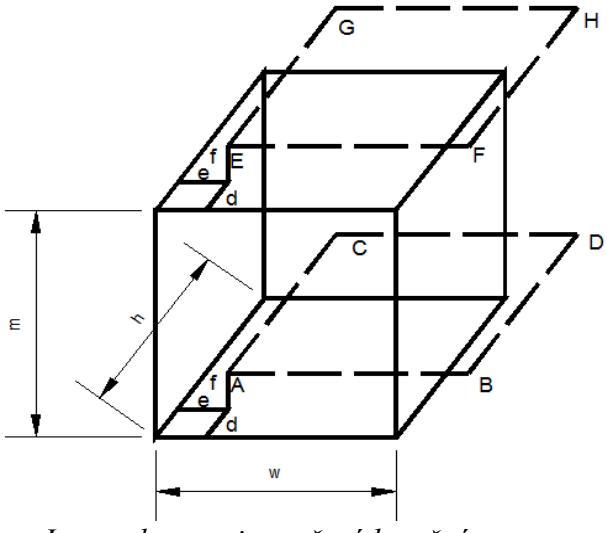
Na obrázku je silnou linkou znázorněná hranice textury. Bod A na textuře získáme interpolací bodů A,B,C a D. Míra interpolace závisí na pozici bodu A uvnitř textury. Vzdálenosti e a d jsou vertikální respektive horizontální vzdálenosti od levé respektive spodní hranice textury. pokud definujeme q jako procentuální vzdálenost část šířky (*e/w*) a p jako (*d/h*), dostaneme hodnotu bodu, který uložíme do textury jako následující rovnici.

$$r = lerp(lerp(Va, Vb, 1-q), lerp(Vc, Vd, 1-q), 1-p)$$

Kde *Va* je hodnota šumové funkce v bodě A a to samé platí pro *Vb*, *Vc* a *Vd*. Funkce lerp je lineární interpolace stejně jako funkce mix v GLSL.

$$lerp(x, y, a) = (1-a)x + a\,y$$

Pokud je A blízko levému dolnímu rohu jsou je hodnota silně ovlivněna hodnotami B,C, a D, čím víc se od rohu vzdalujeme tím víc jejich vliv slábne. Tím dosáhneme toho že jsou hodnoty poblíž pravé hrany blízko hodnotám levé hrany a hodnoty u horní hrany jsou blízké hodnotám dolní hrany. Při skládání této textury není vidět žádný přechod. Popsané vzorce fungují pro dvourozměrnou texturu. U trojrozměrné je to velmi podobné. Jen je třeba interpolovat dvakrát více bodů. Je potřeba interpolovat bod E stejným postupem, akorát použijeme trojrozměrnou funkci šumu. Definujeme s jako (*m/f*)Nakonec interpolujeme body A a E. Vzorec vypadá následovně.



Interpolace trojrozměrné bezešvé textury

$$A = lerp(lerp(Va, Vb, 1-q), lerp(Vc, Vd, 1-q), 1-p)$$
$$E = lerp(lerp(Ve, Vf, 1-q), lerp(Vg, Vh, 1-p), 1-p)$$
$$r = lerp(A, E, 1-s)$$

Tohle všechno jsme dělali proto, že pro mapování trojrozměrné textury použijeme souřadnice vrcholu. Protože vrcholy většiny objektů nejsou v rozsahu od nuly do jedné, dochází téměř vždy k opakování textury. U bezešvé textury to není vidět a je navíc možné snáze ovládat velikost textury.

Implementace

Vertex shader

Klasický vertex shader. Výstupy: Vektory pohledu, světla a normály, umístění vrcholu v pohledových a světových souřadnicích.

Fragment Shader

```
vec4 noiseColor(vec3 Pos, float noiseScale, float ringScale, float frequency, vec3
color1, vec3 color2)
{
   float snoise = 2.0 * texture(NoiseTex, VertexWorld).x - 1.0;
   float ring = fract(frequency * VertexWorld.z + noiseScale * snoise);
   ring *= 4.0 * (1.0 - ring);
   float lrp = pow(ring, ringScale) + snoise;
   vec3  base = mix(color1, color2, lrp);
   return vec4(base, 0.0);
}
```

NoiseColor je funkce, která se stará o vytvoření žilnatého vzoru. Nejdřív vyčteme z textury hodnotu šumu. Pomocí funkce fract, která vrací desetinou část čísla, vytvoříme kruhy. Tím že funkci předáme i hodnotu šumu dosáhneme nepravidelnosti jednotlivých kruhů. Parametry *frequency* a *noiseScale* potom umožňují přesnější kontrolu na vzhledem. *Frequency* určuje četnost kruhů, *noiseScale* určuje míru jakou je tvar kruhů ovlivňován šumem, *ringScale* určuje jaká ze dvou barev bude převládat.
V hlavní funkci potom zavoláme *noiseColor* dvakrát s různými parametry.

Výsledek

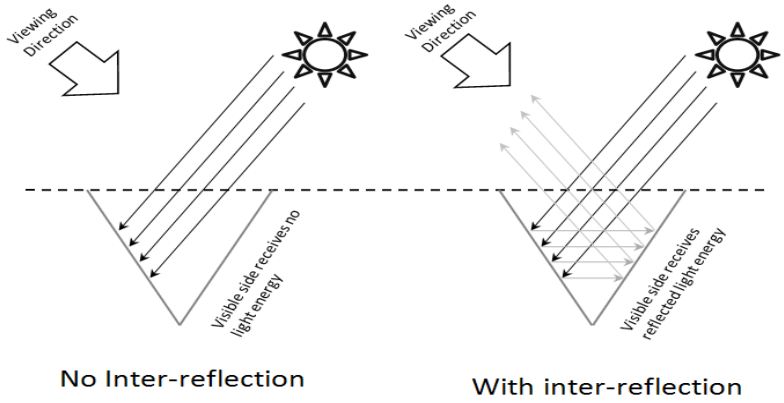
Shader - Mramor

Závěr

Mramoru rozdíl od skla mu chybí průhlednost, ale není pro světlo úplně neprostupný. Mramor je do určité tloušťky průsvitný. Světlo proniká jen pár milimetrů až centimetrů pod povrch mramoru než je kompletně pohlceno, nebo vyzářeno na jiném místě ven. Tomuto jevu se anglicky říká subsurface scattering, což by se dalo přeložit jako pod povrchové rozptýlení. Pro simulaci tohoto jevu ale už nelze použít BRDF (Bidirectional reflectance distribution function) funkci, protože je třeba počítat i s tloušťkou objektu. Implementace této funkce je poměrně náročná a v případě mramoru nemá až takový vliv na výslednou kvalitu a proto jsem se rozhodl ji neimplementovat.

Dřevo

Dřevo je velmi rozšířený materiál, díky svým univerzálním vlastnostem na něj narážíme v běžném životě velmi často. Čisté dřevo je pět dielektrický materiál. Optické vlastnosti dřeva ale velmi závisí na jeho povrchové úpravě. Surové dřevo (nelakované ani neleštěné) má díky struktuře dřeva hrubý povrch a proto použijeme Oren-Nayar stínovací model.
Model Oren-Nayar předpokládá že povrch tělesa je složený z mnoha miniaturních, různě natočených plošek. Oren-Nayar předpokládá že i když ploška viditelná divákem není přímo osvětlena, dochází mezi ploškami k odrazu světla a díky tomu se bude jevit mírně osvětlená, jak je znázorněno na následujícím diagramu.



Výpočet tohoto osvětlovacího modelu je poměrně náročný a proto už sami Oren a Nayar navrhli zjednodušenou formu. V kódu shaderu jsou implementované obě, ale jediný pozorovatelný rozdíl je , že kompletní funkce je při stejném nastavení mírně světlejší. Vzorec pro výpočet kompletní verze, převedený do vektorové formy je následující.

$$I = Diffuse \times Light \times (Normal \bullet Light) \times (C_1[\sigma] + \chi + \epsilon)$$
$$\chi = \cos(\phi_r - \phi_i) \times C_2[\alpha, \beta, \phi_r - \phi_i, \sigma] \times \tan\beta$$

$$\epsilon = (1 - |\gamma|) \times C_3[\alpha, \beta, \sigma] \times \tan\left(\frac{\alpha + \beta}{2}\right)$$

$$C_1 = 1 - 0.5 \times \frac{\sigma^2}{\sigma^2 + 0.33}$$

$$C_2 = \begin{cases} 0.45 \times \frac{\sigma^2}{\sigma^2 + 0.09} \times \sin\alpha & \text{if } \gamma \geq 0 \\ 0.45 \times \frac{\sigma^2}{\sigma^2 + 0.09} \times (\sin\alpha - (\frac{2 \times \beta}{\pi})^3) & \text{otherwise} \end{cases}$$

$$C_3 = \frac{1}{8} \times \left(\frac{\sigma^2}{\sigma^2 + 0.09}\right) \times \left(\frac{4 \times \alpha \times \beta}{\pi^2}\right)^2$$

$$\alpha = max[\arccos(View \bullet Normal), \arccos(Light \bullet Normal)]$$
$$\beta = min[\arccos(View \bullet Normal), \arccos(Light \bullet Normal)]$$
$$\gamma = \cos(\phi_r - \phi_i) \equiv \|View - Normal \times (View \bullet Normal)\| \bullet \|Light - Normal \times (Light \bullet Normal)\|$$

σ – značí hrubost povrchu, je to jediný parametr nastavitelný uživatelem a měl by se pohybovat v rozmezí 0 – 1. Při hrubosti 0 jsou výsledky stejné jako pro Lambertův model.

Implementace

Vertex shader

Je stejný jako ukázkový shader v kapitole Rozbor shaderu, ale navíc předává fragment shaderu pozici vrcholu v objektových souřadnicích.

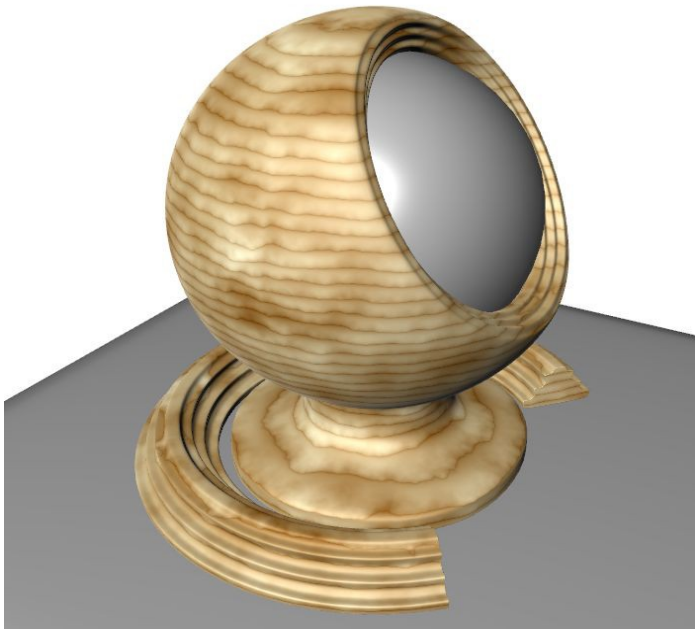
Fragment shader

Pro generování procedurální textury použijeme stejnou funkci jako pro mramor, ale s jinými parametry. Výpočet funkce Oren-Nayar vypadá následovně.

```
float OrenNayarComplete(vec3 L, vec3 E, vec3 N, float Rgh)
{
   float NdotE = dot(N, E);
   float NdotL = dot(N, L);
   float Irradiance = max(0.0, NdotL);
   float AngleViewNormal = acos(NdotE);
   float AngleLightNormal = acos(NdotL);
   float Alpha = max(AngleViewNormal, AngleLightNormal);
   float Beta = min(AngleViewNormal, AngleLightNormal);
   float Gamma = dot(normalize(E - N * NdotE), normalize(L - N *  NdotL));
   float Roughness_2 = Rgh* Rgh;
   float C1 = 1.0 - 0.5 * (Roughness_2/ (Roughness_2 + 0.33));
   float C2 = 0.45 * (Roughness_2/ (Roughness_2 + 0.09));

   if(Gamma >= 0)
   {
      C2 *= sin(Alpha);
   }
   else
   {
      C2 *= (sin(Alpha) - pow((2 * Beta / PI), 3));
   }
   float C3 = 1.0 / 8.0;
   C3 *= (Roughness_2/ (Roughness_2 + 0.09));
   C3 *= pow((4.0 * Alpha * Beta) / (PI * PI), 2);
   float A = Gamma * C2 * tan(Beta);
   float B = (1 - abs(Gamma)) * C3 * tan((Alpha + Beta) / 2.0);

   return max(0.0, NdotL) * (C1 + A + B);

}
```

Zde toho asi není moc co popisovat, jedná se o doslova přepsaný vzorec zmiňovaný výše. Snad jen že autoři v původní práci navrhují při výpočtu C1 nahradit 0,33 hodnotou 0,57 což může přinést mírné zlepšení kvality.

Výsledek



Shader - Dřevo

Závěr

Kvůli vláknité struktuře povrchu nelakovaného dřeva má jeho povrch mírně anisotropní odlesky, které jsem z důvodů jen minimálního dopadu na vzhled neimplementoval.

Barevný lak

Pro jednoduchý barevný lak jsem použil shader Cook-Torrance. Je to první typ stínování, který zavádí pojem microfacets, česky mikroplošky. To jsou miniaturní nerovnosti na povrchu modelu menší než pixel. Je tedy primárně určený pro stínování hrubých povrchů, ale jinak je velmi univerzální a lze ho použít i pro plasty a laky. Shader se řídí pomocí dvou parametrů, hrubosti povrchu a síly odrazu při pohledu zpříma.
Vzorec pro výpočet stínování je následující:

$$R_s = \frac{Fresnel \times Roughness \times Geometric}{(Normal \bullet View) \times (Normal \bullet Light)}$$

Z něj je vidět že výsledek ovlivňují tři části vzorce Fresnel, Roughness, Geometric. Část Fresnel je jasná, je to zjednodušený fresnelův vzorec jako v kapitole o skle. Část roughness určuje počet mikroplošek směřujících k pozorovateli a tím množství odraženého světla. Pro hladký povrch dává podobné výsledky jako Phongův model.

$$Roughness = \frac{1}{m^2 \times (Normal \bullet Half)^4} \times e^{\frac{(Normal \bullet Half)^2 - 1}{m^2 \times (Normal \bullet Half)^2}}$$

Geometric se stará o simulaci situace kdy není osvícena celá část mikroplošky, kvůli velikosti nerovností na povrchu.

$$Geometric = min(1.0, min(\frac{2 \times (Normal \bullet Half) \times (Normal \bullet View)}{View \bullet Half}, \frac{2\text{times}(Normal \bullet Half) \times (Normal \bullet Light)}{View \bullet Half}))$$

Do celku je výsledek rovnice zasazen následovně.

$$R = I \times (Normal \bullet Light) \times (Specular \times R_s + Diffuse)$$

Implementace

Vertex shader

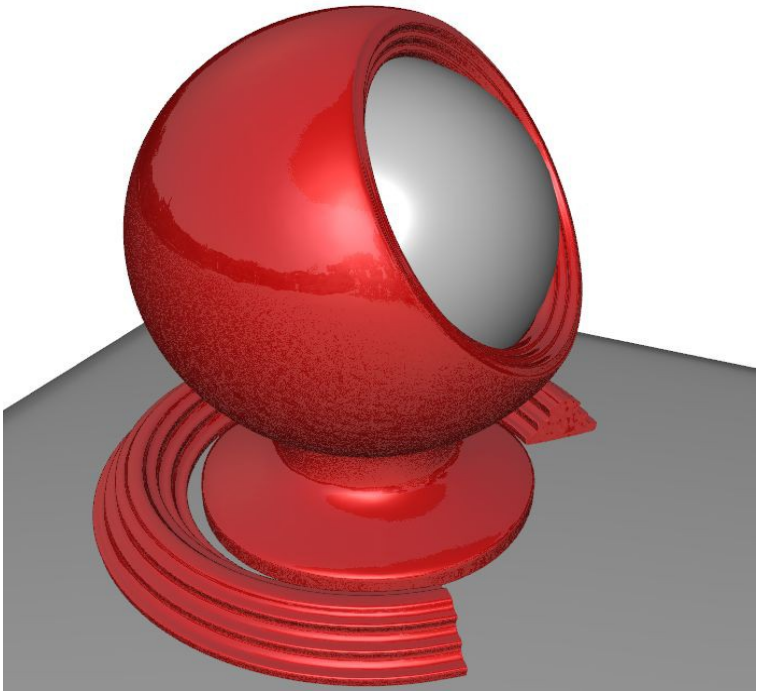
Ve vertex shaderu se dějí obvyklé věci jako převod normál do pohledového prostoru a podobně. Protože jsem ve fragment shaderu použil normálovou mapu a odrazy modifikované normálovou mapou, je třeba poslat do fragment shaderu pár informací navíc. Především inverzní model-view matici pro převod normál z pohledových souřadnic do světových, pro výpočet odrazů z cubemap.

Fragment shader

```
float GeometricTerm(float NdotH, float NdotL, float NdotE, float EdotH)
{
  //geometric term
   float G1 = (2.0 * NdotH * NdotE) / EdotH;
   float G2 = (2.0 * NdotH * NdotL) / EdotH;
   return min(1.0, min(G1, G2));
}

float RoughnessTerm(float NdotH, float RghSQ, bool beckman )
{
   float R = 0.0;
   //roughness
   //beckman
   if(beckman)
   {
      float R1 = 1.0 / (4.0 * RghSQ * pow(NdotH, 4) );
      float R2 = NdotH * NdotH - 1.0;
      float R3 = RghSQ * NdotH * NdotH;
      R = R1 * exp(R2 / R3);
   }
   //gaussian
   else
   {
      float c = 1.0;
      float alpha = acos(NdotH);
      R = c * exp(- (alpha/ RghSQ));
   }
   return R;
}
```

Funkce GeometricTerm je přesným přepisem vzorce. Pro výpočet RoughnesTerm je ale možné použít dva způsoby. Náročnější Beckmannovu metodu nebo výpočetně jednodušší ale méně přesnou Gaussovu.

Výsledek



Shader – Barevný lak

Závěr

Materiál je možné použít jako automobilový lak. Opět se zde nastává stejný problém jako u ostatních shaderů využívajících pro odrazy okolí cubemap a to je absence odrazů ostatních objektů ve scéně.

3.3 Kovy

Měď

Kovy jsou obecně neprůhledné materiály s velkou odrazivostí. Jednotlivé složky, ambientní, difuzní a spekulární mají většinou stejnou barvu. Jejich barvu určuje jakou měrou odráží různé vlnové délky viditelného světla. Nikl a ocel odráží všechny délky s přibližně stejnou intenzitou a mají našedlé odrazy. Na druhou stranu měď a zlato odráží silněji delší vlnové délky a proto mají nažloutlé až načervenalé odrazy.

Stejně jako u skla je pro věrnou simulaci kovů důležitá fresnelova rovnice, zmiňovaná v kapitole o skle. Ovšem zjednodušená forma Shlickova forma je pro kovy hodně nepřesná. Proto jsem se rozhodl použít vzorec navrhovaný Istvánem Lazányi v [3]. Jedná se o upravený Shlickův vzorec, který počítá i s imaginární částí indexu lomu. Tahle část je totiž u kovů výrazně větší než u dielektrik a má větší vliv na optické vlastnosti. Vypadá následovně.

$$F^*(n,k,\cos\theta) := \frac{(n-1)^2 + 4n(1-\cos\theta)^5 + k^2}{(n+1)^2 + k^2}$$

Na vzhledu materiálu taky velmi přidá osvětlení pomocí cubemap.

Implementace

Vertex shader

Ve vertex shaderu se dějí obvyklé věci jako převod normál do pohledového prostoru a podobně. Protože jsem ve fragment shaderu použil normálovou mapu a odrazy modifikované normálovou mapou, je třeba poslat do fragment shaderu pár informací navíc. Především inverzní model-view matici pro převod normál z pohledových souřadnic do světových, pro výpočet odrazů z cube-map.

Fragment shader

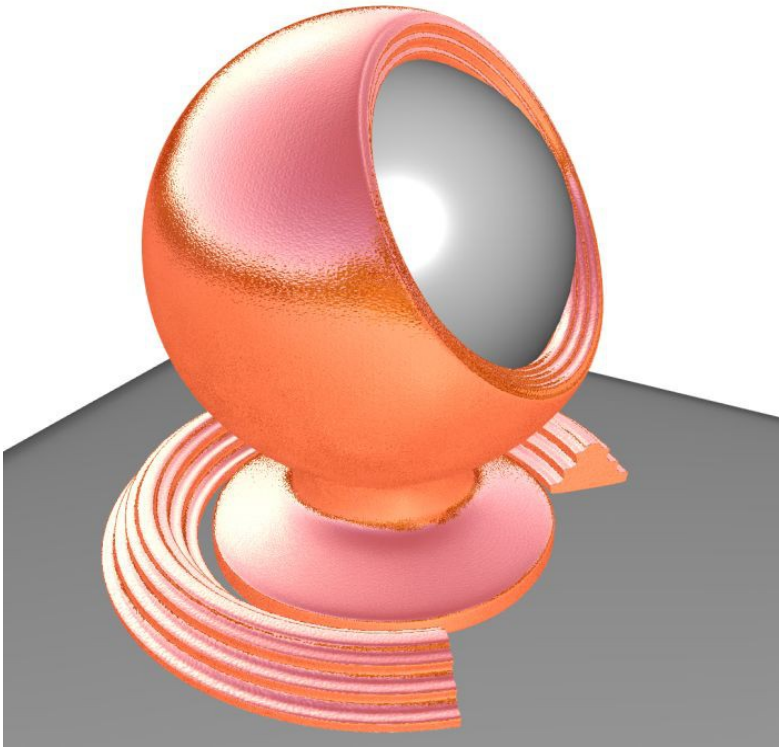
```
float Fresnel2(float n, float k, float NdotE )
{
   float numerator = pow(n - 1, 2.0) + 4 * n * pow(1 - NdotE, 5.0) + pow(k, 2.0);
   float denominator = pow(n + 1, 2.0) + pow(k, 2.0);
   return numerator/denominator;
}
```

Funkce Fresnel2 počítá upravenou Shlickovu rovnici. Parametry jsou index lomu, imaginární část indexu lomu nebo také někdy tzv. extinction coeficient a dot produkt normály a úhlu pohledu.

```
void main(void)
{
  vec3 Nt =  outTBN * normalize(texture(Tex, UV).xyz * 2.0 - 1.0);
  vec3 Ntw = toWorldMat * Nt;
  vec4 ReflectionColor = texture(CubeMapTex, reflect(-EyeVecWorld, Ntw ));
  vec3 E = normalize(EyeVecView.xyz);
  float NdotE = dot(Nt,E);
  out_Color = mix(vec4(ModelColor, 0.0), ReflectionColor * Fresnel2(n, k, NdotE), 0.5);
}
```

Výslednou barvu fragmentu určuje z poloviny barva kovu, a z druhé poloviny barva odrazu. Přičemž intenzitu odrazu určuje výstup z funkce Fresnel2.

Výsledek



Shader - Měď

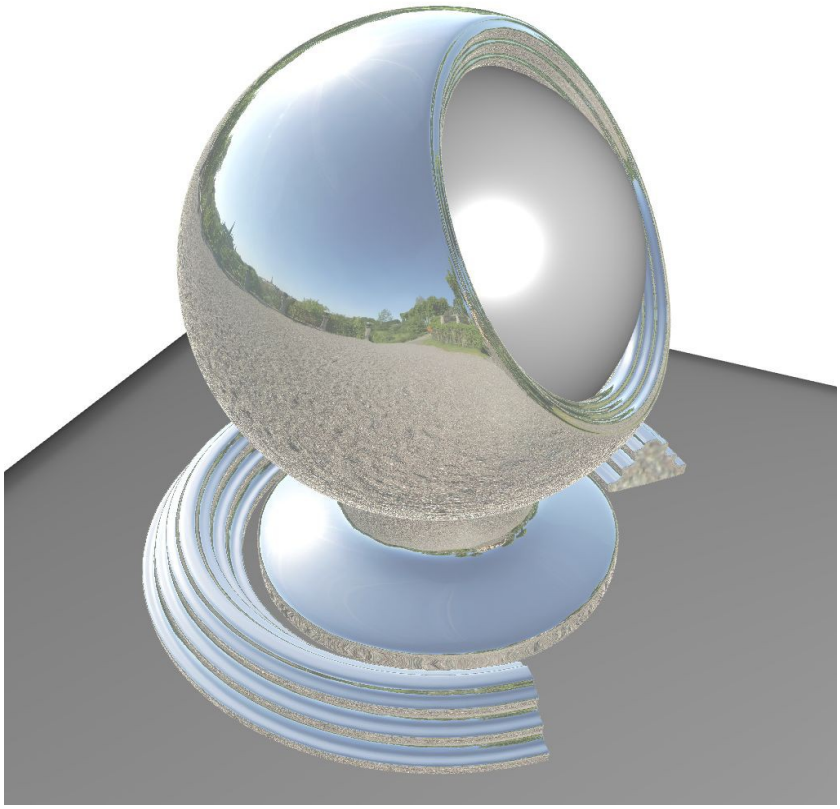
Závěr

Pro dotvoření celkového dojmu kovu jsem použil normálovou mapu hrbolatého povrchu. Výsledek tak připomíná vyleštěnou tepanou měď.

Chrom

Chrom je na tom velmi podobně jako měď. Na rozdíl od ní, ale odráží všechny vlnové délky světla se stejnou intenzitou a nezabarvuje tedy odražený obraz. Pro jeho simulaci se dá použít stejný shader jako pro měď jen s mírnými úpravami a jinými parametry.

Výsledek



Shader - Chrom

Závěr

Poměrně jednoduchý shader spoléhá hlavně na charakteristický zrcadlový odraz chromu, simulovaný pomocí cubemap. Trpí stejným nedostatkem jako sklo, v zrcadlových odlescích se neodráží ostatní objekty ve scéně.

Broušená ocel

Izotropní a anizotropní odlesky

U materiálů s izotropními vlastnostmi mají odlesky ve všech směrech stejnou intenzitu. U anizotropních materiálů ale záleží intenzita odlesku i na struktuře materiálu. Za anizotropii odlesků u materiálu můžou jemné vrypy nebo drážky na jeho povrchu. To způsobí že se odlesky roztáhnou ve směru kolmém na směr drážek.

Materiály s touto vlastností jsou všechny broušené kovy, vlasy a jakýkoliv jiný materiál s opakujícími se nerovnostmi. Simulací tohoto jevu se již zabýval Greg Ward, který také přišel s matematickou rovnicí popisující toto chování materiálů, která vypadá následovně.

$$k_{\text{spec}} = \frac{1}{\sqrt{(N \cdot L)(N \cdot R)}} \frac{N \cdot L}{4\pi\alpha_x\alpha_y} \exp\left[-2\frac{\left(\frac{H \cdot X}{\alpha_x}\right)^2 + \left(\frac{H \cdot Y}{\alpha_y}\right)^2}{1 + (H \cdot N)}\right]$$

N – normála povrchu
L – vektor ke světlu
R – vektor odraženého světla
V – vektor pohledu
H – halfway vektor
X,Y – jsou dva ortogonální vektory v rovině normály, které určují směr anizotropických odlesků.
αx, αy – parametry pro ovládání síly anizotropie v jednotlivých směrech, pokud jsou stejné dostaneme izotropní odlesky.

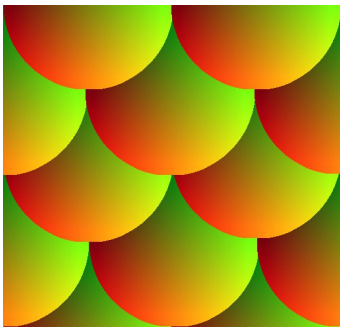
Implementace

Vertex shader

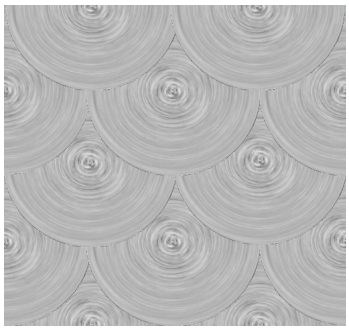
Zde se dějí obvyklé procesy, převod mezi souřadnicovými prostory, výpočet TBN matice a protože nepoužíváme normálovou mapu tak i výpočet vektoru pro cubemap odrazy. Myslím si že uvádět zde kód by bylo jen plýtvání místem.

Fragment shader

```
vec4 Specular(float NdotL, float NdotE, vec3 N, vec3 H)
{
  float FirstTerm = max(NdotL * (1.0 / sqrt(NdotL * NdotE)), 0.05);

  float SecondTerm = max(1.0 / (12.5663706144 * Roughness.x * Roughness.y),0.0);
  vec3 Dir = (texture(Tex, UV) * 2.0 - 1.0).xyz;
  vec3 DirView = normalize(outTBN * Dir);

  vec3 X = normalize(cross(N,DirView));
  vec3 Y = normalize(cross(N, X));

  float HdotX = dot(H, X);
  float HdotY = dot(H, Y);

  float NdotH = max(dot(N, H), 0.0);

  float A = -2.0 * (pow((HdotX / Roughness.x), 2.0) + pow((HdotY/Roughness.y), 2.0));
  float B = 1.0 + NdotH;

  float ThirdTerm = exp(A / B);

  float spec = texture(Tex2, UV).r * ReflectionPower;

  return vec4(LightColor * FirstTerm * SecondTerm * ThirdTerm * spec, 0.0);

}
```



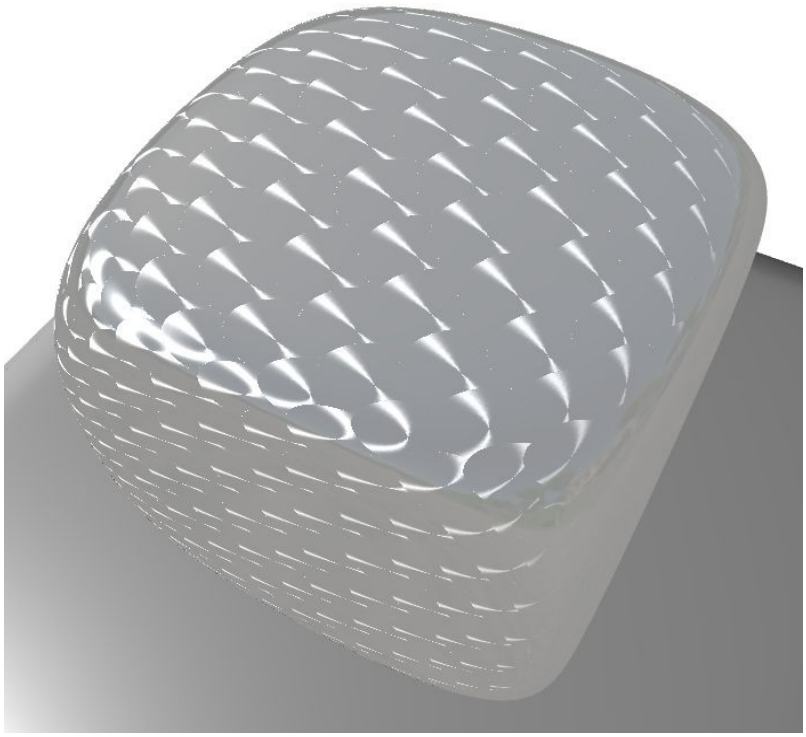
Textura se směrem anisotropních odlesků



Spekulární textura

Funkce Specular je výše zmíněná rovnice převedená do GLSL. Vektory X a Y jsou vyčtené z následující textury. Červená barva reprezentuje velikost komponenty x vektoru a zelená barva y. V každém místě textury směřuje vektor <x,y> ke středu svého kruhu a tím vznikne charakteristický vzhled vybroušených kruhů. Nakonec ještě použijeme texturu k úpravě síly odrazů.

Výsledek



Shader - Broušený kov

Závěr

Výsledek je velmi závislý na správném namapování texturovacích souřadnic tělesa. Vzhledem k tomu že tuto povrchovou úpravu mají spíše plechy a obecně rovné povrchy, jsem se rozhodl použít pro demonstraci krychli.

3.4 Ostatní materiály

Metalický lak

Metalický lak je v dnešní době velmi rozšířený a oblíbený lak především u automobilů. Mezi jeho nejvýraznější vlastnosti patří vysoký lesk a hlavně třpytivý efekt, kterého je docíleno přidáním drobných částeček kovu. Tyto částečky fungují jako malá zrcadla odrážející světlo do různých směrů. To lze dobře simulovat pomocí normálové mapy kde jednotlivé pixely budou znázorňovat odchýlení normál částeček od normály povrchu modelu.

Výsledná barva se skládá ze zrcadlových odlesků způsobenými částečkami kovu, základní barvy a odrazů okolí.

Implementace

Vertex shader

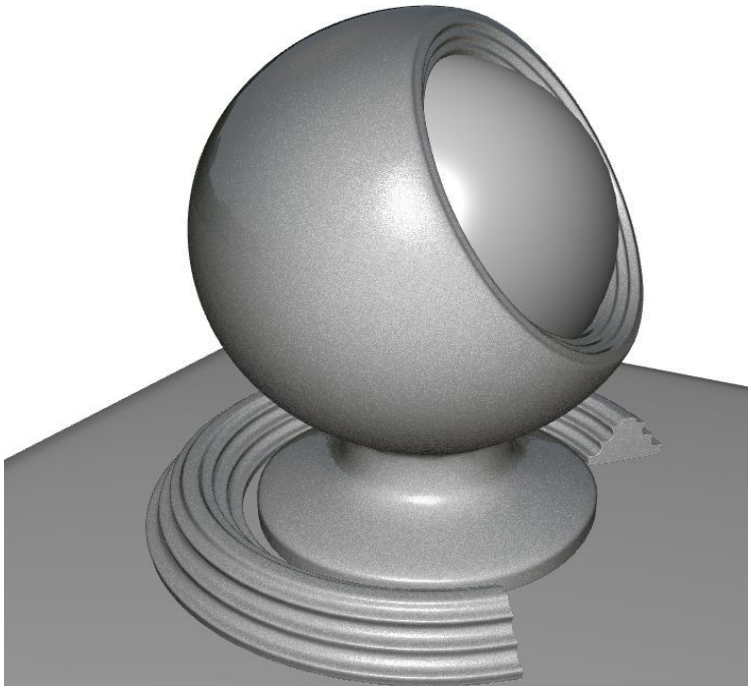
Kód je stejný jako ukázkový vertex shader, ale navíc je přidaná funkce pro výpočet TBN matice a výpočet vzdálenosti pozorovatele od objektu.

Fragment Shader

Zde je nejdůležitější funkce Sparkle která se stará o charakteristické odlesky.

```
vec4 Sparkle(vec3 N, vec3 E, vec3 H, vec3 Tnorm)
{
  vec3 TexNormalView = TBNview * Tnorm;
  vec3 VNp1 = 0.1 * Tnorm + 1.0 * N;
  vec3 VNp2 = 1.0 * (Tnorm + N);
  VNp1 = normalize(VNp1);
  VNp2 = normalize(VNp2);
  float  Lambert1 = max( dot( VNp1, E), 0.0);
  float  Lambert2 = max( dot( VNp2, E), 0.0);

  vec3 SparkleColor = ColorSparkle * Lambert1 + pow(Lambert2, 16) * ColorSparkle;

  float NdotH = max(0.0, dot(H, N));
  vec3 FinalSparkle = SparkleColor * Attenuation + SparkleColor * pow(NdotH, 80.0) +
SparkleColor * 0.4;
  return vec4(FinalSparkle, 0.0);
}
```

Výsledek



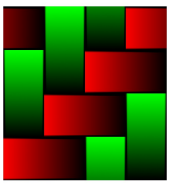
Shader - Metalický lak

Závěr

Výsledek velmi připomíná stříbrný metalický lak používaný v automobilovém průmyslu. Simulace je nejvěrnější při pohledu z menší vzdálenosti a při silnějším osvětlení.

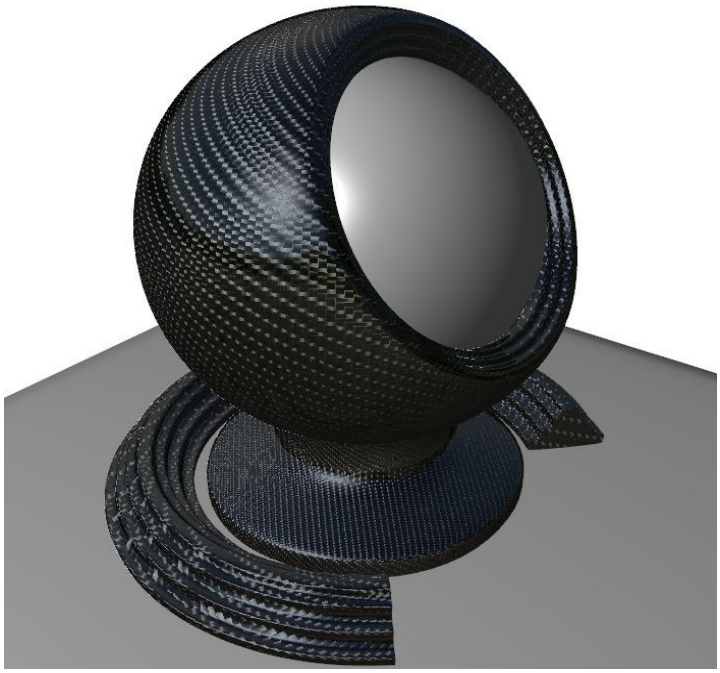
Uhlíková vlákna

Materiály které se nehodí ani do kategorie dielektrik ani kovů. Překvapivě není těchto materiálů zrovna málo. Příkladem mohou být například uhlíková vlákna zalitá v epoxidové pryskyřici. Tento materiál se používá k výrobě rámů bicyklů nebo na tuningové prvky na karosériích automobilů. Přesto že uhlík je kov má tento materiál díky epoxidu chování spíše dielektrického materiálu. Díky zarovnání uhlíkových vláken má anizotropní odlesky. A díky vrstvě epoxidu i výrazné zrcadlové odlesky. Pro materiál jsem použil shader podobný jako pro leštěnou ocel, kvůli anisotropním odleskům. Vzor, který jsem použil pro řízení směru anisotropních odlesků je na obrázku.



Textura se směry anisotropie

Výsledek



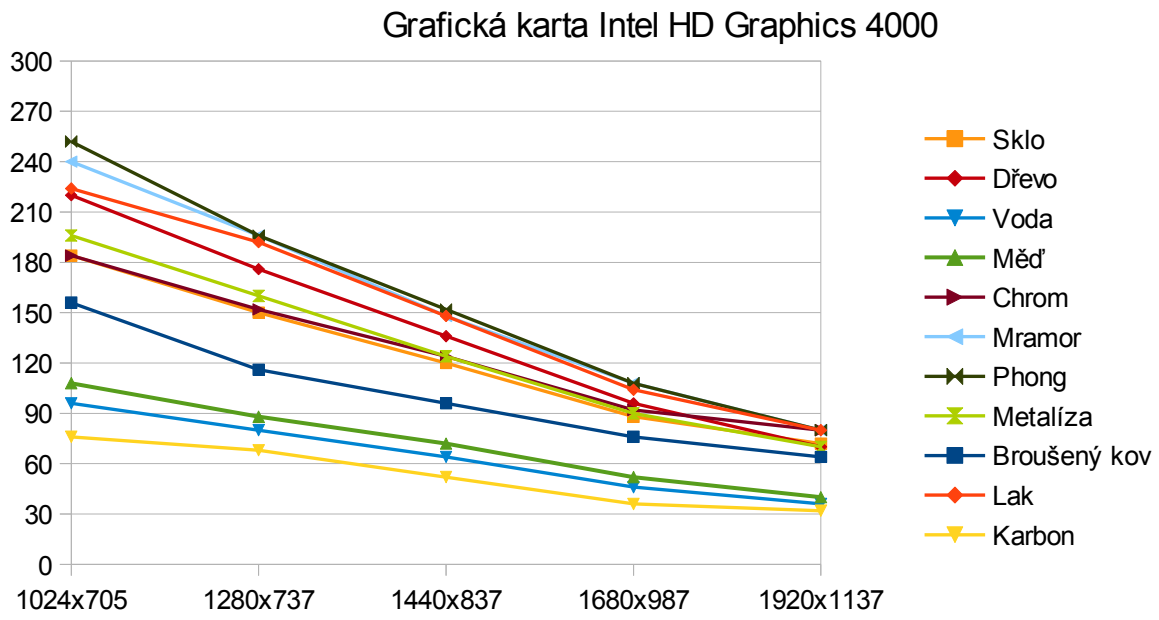
Shader – Uhlíková vlákna

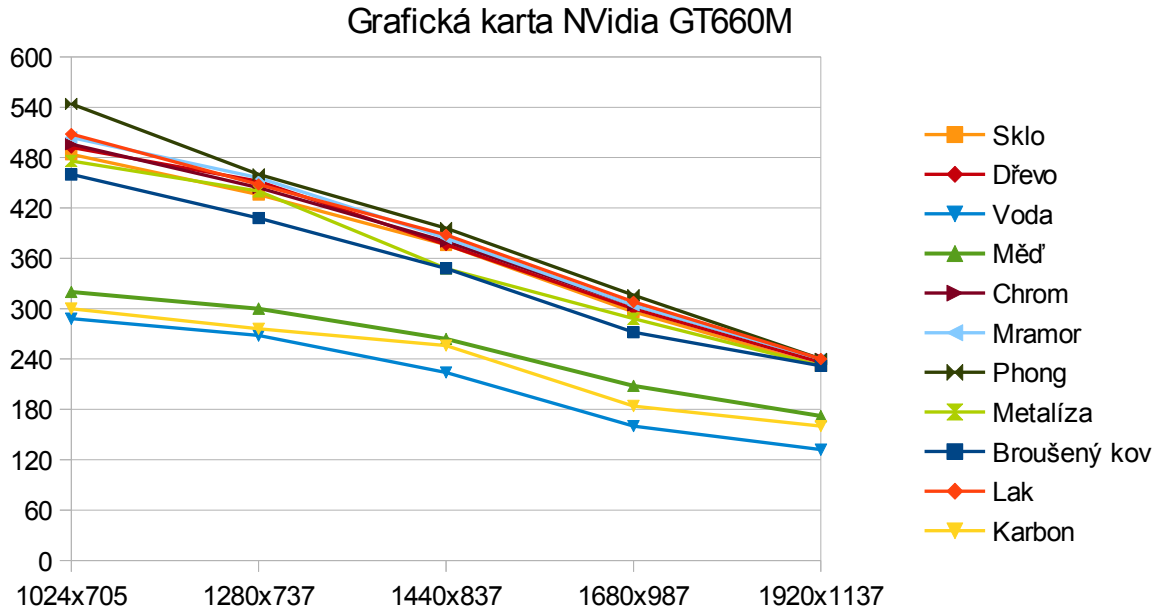
Závěr

Shader má poměrně velikou výpočetní náročnost způsobená poměrně náročným výpočtem anisotropních odlesků a velkým opakováním textury. Výpočetní nároky by se daly snížit použitím textury s nižším rozlišením.

4 Srovnání výkonu shaderů

Výkon jsem srovnával následujícím způsobem. V okně jsem nechal rotovat krychli zvětšenou na velikost okna, spolu s objektem znázorňujícím světlo a odečítal jsem počet snímků za vteřinu. Tento postup jsem opakoval na pěti různých rozlišeních a dvou grafických kartách. Výsledky jsou zanesené v grafu. Rozlišení jsou nestandardní, protože program běžel v okně.





Z výsledků vyplývá že nejnáročnější shadery jsou voda a překvapivě karbon. U vody je to způsobené výpočtem šumu v průběhu vykreslování. U karbonu na vině poměrně složitý výpočet anisotropních odlesků a hustě namapované textuře se směry anisotropie. Naopak nejméně náročný je shader jednobarevného laku spolu s mramorem. Zajímavé je že na obou kartách dopadly nejlépe a nejhůře rozdílné shadery.

5 Závěr

Cílem práce bylo navrhnout metody a implementaci pokročilých stínovacích programů. Toto zadání jsem splnil v kapitole o implementaci shaderů, kde probírám metody simulace zvolených materiálů. Přínos této práce spočívá také v představení nejnovější specifikace OpenGL a GLSL, která se značně liší od verzí před OpenGL 3.3 jak na straně Windows aplikace tak v shaderové části.
Samozřejmě je ještě spoustu prostoru k vylepšování. Nedostatky jednotlivých shaderů jsou zhodnoceny v závěru každé kapitoly, ale jeden problém se vyskytuje téměř u všech shaderů s odrazy okolí a to je absence ostatních těles v odrazech. Zde vidím nejvíce potenciálu ke zlepšení vizuální stránky shaderů.

6 Seznam použité literatury:

[1] – David Wolf - OpenGL 4.0 shading language cookbook, 2011

[2] - http://www.opengl.org/sdk/docs/manglsl/ - OpenGL Shading Language (GLSL) Reference Pages

[3] – István Lazányi – Fresnel term approximations for metals, 2005

[4] - Wolfgang F. Engel - Shaderx2 - Shader Programming Tips & Tricks With Directx 9, 2003

[5] - http://www.lighthouse3d.com/tutorials/

7 Přílohy

Ovládání programu

Levé tlačítko myši	–	rotace kamery kolem předmětu
Pravé tlačítko myši	–	posun centra rotace
Num0	–	zastaví rotaci světla
Num2 a Num8	–	rotace kamery po ose X
Num4 a Num6	–	rotace kamery po ose Y
Num5	–	vrátí kameru do výchozí polohy
Escape	–	ukončí program

Program pro spuštění vyžaduje Visual C++ Redistributable pro Visual Studio 2012, které je možné stáhnout na stránkách Microsoftu. Pro běh programu je třeba grafická karta s podporou OpenGL alespoň ve verzi 3.3.

Popis formátu spouštěcího souboru.

Program se spouští pomocí textového souboru, který obsahuje cesty k souborům textur, 3d objektům, kódům shaderů a uniform proměnné. Formát souboru je následující.

 # - značí komentář, čtení souboru pokračuje na dalším řádku

Vertex shader
<V>[cesta(absolutní nebo relativní)]
Fragment shader
<F>[cesta(absolutní nebo relativní)]
3D objekt
<O>[jméno];[cesta(absolutní nebo relativní)];[pořadí přiřazeného shaderu];
Textura 2D/Cubemap
<U>[jméno];[T];[cesta(absolutní nebo relativní)];[T/F];
Vektor
<U>[jméno];[V];[float];[float];[float];
Float
<U>[jméno];[F];[float];
Textura šumu 3D/2D
<U>[jméno];[N];[rozměr textury(pix)];[T/F];

- Vertex a fragment shadery jsou párované na základě pořadí přidání.
- Relativní cesta je vztažená k umístění spouštěcího souboru.
- Pořadí shaderu je pořadí ve spouštěcím souboru a začíná nulou.
- Poslední parametr u textury určuje zda se jedná o cubemap nebo normální texturu, kde T značí cubemap a v tomto případě se jako cesta uvádí pouze cesta ke složce s texturami, ve které je umístěno šest obrázků s následujícím pojmenováním *X_pos.png, X_neg.png, Y_pos.png, Y_neg.png, Z_pos.png* a *Z_neg.png*.
- Pro texturu šumu určuje poslední parametr T nebo F zda se má vygenerovat trojrozměrná nebo klasická textura, T značí trojrozměrnou.
- Jména všech proměnných se musí shodovat se jmény uvedenými v shaderech aby mohly být správně přiřazeny.
- Proměnné jsou pro všechny shadery společné. Jako první objekt je třeba uvést krychli pro zobrazení cubemap a pojmenovat jí „*SkyBox*“, objekt světla se musí jmenovat „*Svetlo*“ obojí bez uvozovek.
- Podporované formáty textur jsou: png, jpg, bmp.
- Podporované formáty 3D objektů jsou: 3ds, dae, obj, ply, stl.

Obsah CD

/Src – zdrojové kódy k aplikaci

/Bin – zkompilovaná aplikace spolu se spouštěcími soubory a daty potřebnými k běhu

/Doc – text práce ve formátu pdf